DETERMINING SOFTWARE COMPLEXITY

FIELD OF THE INVENTION

{001} The invention relates to the field of software engineering, and more particularly to methods, apparatus, and computer program products for determining software complexity.

BACKGROUND

{002} Software has become increasingly complex as processor capability, memory density, and users' expectations have grown. As a result, methods and tools for managing software development projects have become increasingly important, including methods for determining software complexity to be used in estimating, for example, how many defects are expected to occur in a software component, how many hours of development time are expected to be needed for the completion of a project, and so forth.

{003} Today, such estimates are normally based on counts of lines of code, together with some simple rules for determining what, roughly, constitutes a line of code. For example, a certain development time and a specified number of defects may be expected per thousand lines of code. This method may be called generically the KLOC method.

{004} The KLOC method, while certainly useful, has significant drawbacks. These drawbacks are a product of the highly variable nature of software components. Some components are rich in unique code, whereas other components include substantial repetitions, spaces, blank lines, comments, and so forth. Thus, when two software components are compared using the KLOC method, where one component is rich in unique code while the other is highly repetitive and full of comments, the resulting estimates will be inconsistent. The two estimates might be numerically the same, for example, whereas in reality the software that is rich in unique code is rationally expected to be more difficult to develop, and therefore to require more development time and be more susceptible to defects. Furthermore, the KLOC method is strongly tied to the properties of the particular programming language in question, as some languages are inherently more dense than others.

{005} Thus, there is a need for a language-independent way to determine software complexity consistently, so that software project estimates such as expected development time, expected numbers of defects, and so forth, may be determined more accurately than is possible today.

SUMMARY

{006} Embodiments of the invention include methods, apparatus, and computer program products for determining software complexity. A plurality of versions of a software module whose complexity is to be determined are compressed. Lengths of the compressed versions are compared, one with another, to provide complexity metrics.

BRIEF DESCRIPTION OF THE DRAWINGS

{007} FIG. 1 is a flowchart that illustrates an exemplary method for providing program complexity metrics according to the present invention.

{008} FIG. 2 is an illustrative embodiment of apparatus according to the present invention.

DETAILED DESCRIPTION

{009} The present invention includes language-independent methods, apparatus, and computer program products for determining software complexity more accurately and consistently than is possible using the KLOC method.

{010} Measures are taken of a plurality of different forms of a software component whose complexity is to be determined, and the measures are then compared with one another to reveal characteristics of the software component that are otherwise obscured. More particularly, a plurality of versions of the software are determined, each of the versions is compressed, and the lengths of the compressed versions are compared with each other to provide software complexity metrics.

{011} As an aid to understanding the invention, let an exemplary software module M be constructed from three strings, which are called here p, p', and p''. Let $K(x)$ be the KLOC measure of the complexity of string x. The complexity of the module M would then be the sum of the lengths of the three strings, i.e., $K(M) = K(p) + K(p') + K(p'')$.

{012} Suppose, however, that the strings are not independent, but rather that p' is dependent upon p, i.e., $p'=f(p)$, and p'' is dependent upon p and p', i.e., $p''=g(p, f(p))$. When $f(.)$ and $g(.)$ are relatively simple functions, for example substitutions of identifiers, it is more reasonable and more useful for purposes such as estimating the number of defects in the module, to take into account conditional dependencies to represent the incremental contributions of p' and p''. Thus, a complexity measure according to the present invention, which is called here $C(M)$, may be described in terms of the complexity of p, of p' given p, and of p'' given p and p', i.e., $C(M) = C(p) + C(p'|p) + C(p''|p, p')$.

{013} Turning now to a preferred embodiment of the invention, which may be understood in the theoretical context just described and with reference to FIG. 1, let *P0* be the raw program text of *P*, let *P1* be the normalized program text of *P*, and let *P2* be the normalized unique program text of *P*. Here, the raw text *P0* is found by collecting the program files of *P* into one file. In a preferred embodiment of the invention, the normalized program text *P1* is found by eliminating comments from *P0*, normalizing sequences of spaces into a single space, and then sorting the remaining lines into lexicographic order. This way of normalizing the program text is merely illustrative of the invention rather than limiting, however, as there are many other ways to normalize, all of which fall within the scope of the invention. In another exemplary embodiment, the normalized program text *P1* may be found by reformatting the program text *P0* according to a stylistic standard, so that minor differences in formatting style are removed. This approach may be especially useful when the software in question has a long life, as style fashions tend to evolve over time. The normalized unique program text *P2* may be found by eliminating duplicate lines in *P1*.

{014} Operations of a corresponding method are shown in FIG. 1. From *P*, the raw program text *P0* is determined (step 100), the normalized program text *P1* is determined (step 110), and the normalized unique program text *P2* is determined (step 120), all as just described.

{015} Texts *P0*, *P1*, and *P2* are then compressed (step 130). In a preferred embodiment of the invention, compression is provided by application of the open source bzip2 program, for example version 1.0.1 of bzip2. The use of this particular compression algorithm is merely illustrative of the invention rather than limiting. The bzip2 compression method, which relies on a block sorting algorithm and numeric coding, is well known to those skilled in the art, and therefore will not be described in detail here. Further information regarding bzip2 may be found on the World Wide Web at, for example, Uniform Resource Locator digistar.com/bzip2/.

{016} Measures *C0, C1,* and *C2* are then found from the compressed versions of *P0*, *P1*, and *P2*, respectively (steps 140, 150, 160). Measure *C0* is the length of the compressed version of *P0*. Measure *C1* is the length of the compressed version of *P1*. Measure *C2* is the length of the compressed version of *P2*. The resulting measures *C0*, *C1*, and *C2* are compared by computing the ratios *C0/C1* and *C1/C2* (step 170).

{017} Measure *C0*, which results from compression of the raw program text, may be used rather than a KLOC count in estimates of expected development times and expected numbers of defects. Measures *C1* and *C2* address the question of incremental contributions. Thus, the ratios *C0/C1* and *C1/C2* are proportional to the redundancy of the implementation of *P* and the propagation of defects, respectively, and may be used as metrics of these attributes.

{018} As shown in FIG. 2, apparatus according to the present invention includes logic 200, which may itself include memory (not shown), a compressor 210, and a divider 220. These elements are shown as separate in FIG. 2 only for descriptive convenience. All may be implemented using a stored-program-control processor, such as a microprocessor.

{019} The logic 200 determines the raw program text *P0*, the normalized program text *P1,* and the normalized unique program text *P2* as described above. The compressor 210 compresses the texts *P0*, *P1*, and *P2*. In a preferred embodiment, the compressor uses release 1.0.1 of bzip2. The logic 200 determines the measures *C0*, *C1*, and *C2*, which are, respectively, the lengths of the compressed versions of *P0*, *P1*, and *P2*. The divider 220 computes the ratios *C0/C1* and *C1/C2.*

{020} Embodiments of the invention further include program storage devices readable by machines, tangibly embodying programs of instructions suitable for implementing the methods described above and for controlling processor implementations of the apparatus described above.

{021} Thus, as described above, the present invention provides language-independent methods, apparatus, and computer program products for determining software complexity metrics that are more accurate and consistent than measures based upon the KLOC method. The foregoing description of the invention is illustrative rather than limiting, however, and the invention is limited in its scope only by the claims appended here.